TP3: Search Engine

Alef Henrique de Castro Monteiro

3 de julho de 2017

1 Introdução

Nos mecanismos de busca, a forma de indexação e o modelo de ranqueamento utilizado para processar a consulta tem papel fundamental na eficiência da recuperação da informação. Entre as formas de indexar os termos e os documentos da coleção, está o Índice Invertido (ou Arquivo Invertido). A utilização de arquivo invertido aumenta a eficiência de pesquisa em várias ordens de magnitude e, por isso, sua implementação se tornou essencial para as máquinas de busca. Em relação aos modelos para o cálculo do ranking, temos várias opções: Modelo Vetorial, Modelo Probabilístico, Modelo Booleano, entre outros.

Motivado pelo desafio de realizar busca eficiente em uma coleção com milhões de páginas da Web, o Trabalho Prático 3 tem por objetivo a implementação de um Processador de Consultas a partir de diferentes modelos de ranqueamento. Neste relatório, são apresentados os detalhes da implementação do Processador de Consultas, bem como as decisões de design que influenciaram o desempenho da busca e dos documentos. Por fim, são realizados diversos experimentos que verificam a influência que alguns fatores têm na eficiência e velocidade da indexação.

2 Definição do Problema

O problema abordado pelo trabalho é projetar e implementar um programa que seja capaz de realizar a busca de termos em uma coleção estática com milhões de documentos da web. O objetivo é recuperar eficientemente informações contidas nesses grandes arquivos armazenados em memória secundária com a utilização de diferentes algoritmos de busca.

Para isso, uma coleção com cerca de 6,2 milhões de páginas em HTML deve ser indexada em um tipo de índice conhecido como arquivo invertido. Assim, a partir desse conjunto de documentos, deve ser criado um arquivo invertido constituído de uma lista ordenada de palavras-chave, onde cada palavra-chave tem uma lista de apontadores para os documentos que contêm aquela palavra, bem como a frequência desta palavra no documento. Além da criação do índice invertido em relação aos termos da coleção, deve ser criado também um segundo índice invertido em relação aos *anchor texts*, para que seja possível utilizá-los nos algoritmos de busca.

Além da indexação, deve ser implementado um processador de consultas que permita ao usuário realizar a busca com pelo menos três modelos de ranking diferentes: Vector Model, Vector Model com Anchor texts e Vector Model com PageRank. Para facilitar a interação entre o usuário e o programa, deve ser criada também uma interface em HTML que possibilite a realização da busca e a visualização dos resultados retornados pelo programa.

3 Solução do problema

Para solucionar o problema proposto pelo trabalho, foram implementados dois programas em *C++*: o Indexador(*Indexer*) e o Processador de consultas(*Searcher*). No Indexador, além das bibliotecas padrões da linguagem, o programa utiliza também uma parte da biblioteca externa **Gumbo**[1] para fazer o *parsing* em cada

[1]Biblioteca Gumbo Parser C++ disponível em `https://github.com/google/gumbo-parser`

documento HTML. Então, para facilitar a implementação e ajudar na eficiência do algoritmo, o programa utiliza a biblioteca para extrair o texto puro da página HTML. Já o Processador, por sua vez, utiliza como auxílio um servidor em *C++*[2] para realizar a interação entre a interface em HTML e o programa implementado em *C++*.

Na **seção 2** foi apresentada a definição do problema proposto para este trabalho. Neste momento, serão apresentados os detalhes da implementação do programa utilizado no trabalho, assim como as decisões de design que foram tomadas para respeitar as condições impostas pelo trabalho. Para isso, esta seção foi dividida em duas partes principais que detalham cada um dos programas: *Indexador e Processador de Consultas*.

3.1 Indexador

Como é necessário indexar também os *anchor texts*, para realizar a indexação o programa realiza duas passagens pela coleção. Na primeira, o programa extrai o conteúdo das páginas e realiza a indexação dos termos. Na segunda, ele extrai os links presentes nas páginas para realizar a indexação dos *anchor texts* e o cálculo do PageRank das páginas da coleção. O motivo de se realizar esta segunda passagem é porque o conhecimento dos IDs dos documentos é necessário para identificar o documento correspondente ao link extraído de uma certa página. Durante cada um desses dois processos, o Indexador realiza quatro tarefas: *Parsing, Mapeamento, Indexação e PageRank*. A seguir, a implementação de cada uma delas será detalhada.

3.1.1 Parsing

A fase de *parsing* começa com a separação das páginas em HTML, que inicialmente encontram-se agrupadas em vários arquivos. Cada um desses arquivos contém um número variado de páginas HTML, conforme o modelo a seguir:

$|||<URL>|<HTML>|||.....|||<URL>|<HTML>|||$

Então, o programa abre cada um dos arquivos da coleção, processa as páginas uma a uma, retirando apenas a url e o texto presente no HTML de cada uma e efetuando a indexação dos termos da página antes de prosseguir para a página seguinte. A cada página HTML lida o programa cria um identificador *docID* para a página, que é armazenado em uma tabela Hash onde a chave é a URL da página.

Para a obtenção do texto contido no HTML, o programa utiliza a função *get_cleantext()* presente na biblioteca Gumbo. Obtido o texto, o programa coloca todo o texto em letras minúsculas apenas (*lowercase*) e utiliza o método *remove_accents()* para retirar os acentos e os caracteres especiais presentes. Todo esse processo pode ser visto no algoritmo 1, que exemplifica o fluxo do programa para realização do *parsing*. Assim, após a obtenção do texto limpo, pode-se realizar o mapeamento dos termos.

No entanto, durante a segunda passagem pela coleção o processo é um pouco diferente. Nesse momento, em vez de extrair o texto da página o programa utiliza a função *search_for_links* e extrai apenas os textos que estão dentro das tags $< a >$ (*anchor texts*). Dessa forma, é feito o mapeamento dos termos para cada um dos *anchor texts*. Além de realizar o mapeamento dos termos, nesta etapa o programa cria as arestas para o grafo das páginas que será utilizado para calcular o PageRank. Para representar esse grafo, são utilizadas listas de adjacência com a estrutura $vector < vector < int >>$. Assim, ao criar essas arestas, o programa verifica o ID da páginas onde sai o link e o adiciona à lista da página correspondente ao link, ou seja, a página que recebe o

[2]Eidhein Simple-Web-Server disponível em `https://github.com/eidheim/Simple-Web-Server`

link.

Algoritmo 1: Parsing

Entrada: Hash table *documents*, File f

```
1  string html, url
2  extract_html(f, url, html)
3  while html! = NULL do
4  |  id ← documents.size()
5  |  documents[url] ← id
6  |  get_cleantext(html)
7  |  lowercase(html)
8  |  remove_accents(html)
9  |  extract_html(f, url, html)
10 end
```

3.1.2 Mapeamento

Para o mapeamento dos termos de cada texto, o programa cria uma classe *Mapper*, que é responsável por identificar a ocorrência de cada um dos termos presentes na coleção e mapeá-las em forma de tuplas. A ideia é que para cada termo identificado no texto seja criada uma tupla que armazena o ID do termo, o ID do documento correspondente à página HTML que está sendo processada, a frequência do termo neste documento e a posição na qual o termo está localizado no documento. Ou seja, cada tupla tem o formato $< termId, docId, freq, pos >$. Para realizar todo este processo de mapeamento, foi criado o principal método desta classe, que é o *extract_triples()* para o mapeamento do texto e o *extract_anchortext_triples()* para o mapeamento dos *anchor texts*.

Inicialmente, o programa cria uma instanciação da classe *Mapper* e, por meio do construtor da classe, armazena as stopwords[3] em um vetor. Após essa etapa, a primeira tarefa é separar as palavras do texto em *tokens*. Assim, o programa faz a leitura do texto e para cada *token* é feita a verificação se esse termo pode ou não ser indexado. Neste trabalho, os critérios utilizados para indexar um termo são os seguintes:

- Não estar na lista de stopwords.
- Ter comprimento entre 2 e 30 letras inclusive
- Não conter números

Apesar de números poderem ser extremamente relevantes, como é o caso da busca por datas, horários, leis, etc, neste trabalho eles são excluídos devido ao limite de memória do programa ser baixo. Além disso, palavras muito grandes também são ignoradas para evitar a indexação daquelas que eventualmente tenham sido concatenadas por falhas durante o *parsing*. Já com a finalidade de melhorar a eficiência durante o processo de busca, as stopwords não são indexadas para que o processador de consultas booleano retorne a mínima quantidade possível de documentos irrelevantes para uma dada consulta.

No mapeamento do texto das páginas da coleção, caso o termo possa ser indexado e ainda não esteja no vocabulário do programa, ele é adicionado neste. Para representar o vocabulário, o programa utiliza um "unordered_map"que mapeia cada termo e seu respectivo número identificador $< termo, termId >$ em uma tabela Hash. Depois de adicionar o termo ao vocabulário, o programa cria a tupla $< termId, docId, freq, pos >$ e a adiciona à *run* do Mapper. No entanto, como o programa escreve as tuplas na medida em que lê o texto, não é possível saber qual o valor da frequência do termo naquele documento ($freq$). Para solucionar este problema, o programa escreve na tupla a frequência do termo até aquele momento e, ao gerar o índice final, o programa substitui cada um desses valores pela frequência correta. Cada *run*, por sua vez, corresponde à lista de tuplas encontradas até o momento e tem seu tamanho limitado à 1.000.000 de tuplas.

[3]Lista de stopwords em Português e em Inglês obtida em "http://www.ranks.nl/stopwords/", às 12:08h do dia 15/05/17

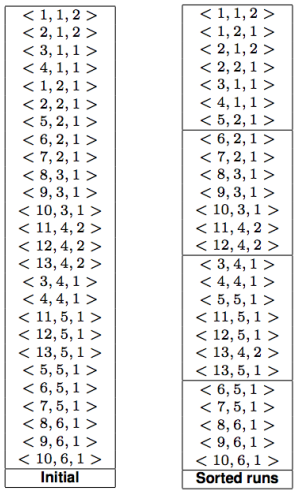

Figura 1: Separação das tuplas em *runs* ordenadas que são armazenadas em diferentes arquivos

Caso atinja o seu tamanho máximo, o programa abre um arquivo temporário e armazena a *run* em disco. Assim como mostrado na Figura 1, cada *run* é ordenada antes de ser armazenada no arquivo temporário. Dessa forma, todo esse processo é repetido até que tenham sido processados todos os arquivos da coleção. Assim, após a conclusão do mapeamento dos termos e dos documentos da coleção, existem vários arquivos temporários que com as tuplas obtidas durante o processamento dos arquivos. Enfim, todo o processo de mapeamento executado pelo programa pode ser representado pelo algoritmo a seguir.

Algoritmo 2: Extract_triples

```
   Entrada: string text, vector of triples run, available memory M
1  for word in text do
2      if !indexable(word) then
3          //Go to next iteration
4          continue
5      end
6      else
7          add_vocabulary(word)
8      end
9
10     termId ← vocabulary[word]
11     term_freq[termId] ← term_freq[termId] + 1
12     freq ← term_freq[termId]
13     t ← triple(termId, docId, freq, pos)
14     run.push_back(t)
15     k ← (M − run.size())/sizeof(t)
16     if run.size() >= k then
17         store_run(run)
18         run.clear()
19     end
20 end
```

Algoritmo 3: Mapping

```
1  for each Document docId do
2      Mapper.extract_triples(docId)
3      // Store last run
4      Mapper.store_run(Mapper.run)
5  end
```

3.1.3 Indexação

Após a etapa de mapeamento dos termos, tanto do texto quanto dos *anchor texts*, é necessário gerar o índice final que contém o arquivo invertido. No entanto, conforme a Figura 2, é necessário juntar todas as *runs* e gerar uma única lista contendo todas as tuplas ordenadas, ou seja, é necessário realizar um *merging* entre as *runs*.

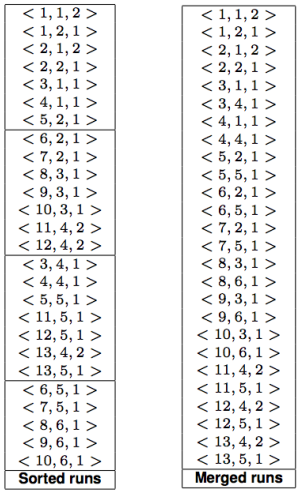

Figura 2: Merging das *runs* ordenadas

Para isso, o programa cria uma Heap com o tamanho do número de *runs* geradas e insere nela a primeira tupla de cada um dos arquivos. Como critério de ordenação, a heap utiliza o menor *termId* e, caso o termo seja o mesmo, prevalece o menor *docId*. Dessa forma, a heap é ordenada e o primeiro elemento é removido e adicionado à *run* de saída, que armazenará as tuplas enquanto houver memória disponível. Assim como no mapeamento, caso a *run* atinja o tamanho máximo da memória disponível, ela é armazenada em disco. A diferença é que neste momento as *runs* são armazenadas no mesmo arquivo, uma após a outra. Após a remoção da tupla na heap, a próxima tupla presente no arquivo que continha a tupla retirada é adicionada à heap. Caso não haja mais tuplas em um arquivo, este é excluído do disco e a heap tem seu tamanho reduzido em uma unidade. Na sequência, a heap é ordenada novamente e o processo se repete até que a heap fique vazia. Para realizar todo esse processo, foi criada a classe *Indexer*.

Ao concluir o *merging* das *runs*, a única tarefa que programa deve executar é a atualização do valor das frequências $freq$ nas tuplas dos índices. Dessa forma, o programa percorre cada linha do índice gerado pelo *merging* das *runs* até que a tupla contenha um par $(termId, docId)$ diferente. Dessa forma, cada uma das tuplas é armazenada em um buffer de tuplas, onde o programa verifica o valor máximo de $freq$ para o par ($termId$, $docId$), que corresponde à correta frequência do termo naquele documento. Assim, o programa reescreve o bloco de tuplas com os valores corretos em um arquivo separado. Além disso, ao reescrever o bloco, o programa armazena em uma tabela hash a posição (em bytes) onde cada bloco foi armazenado. Com isso, o processador de consultas poderá ler ler apenas a parte do índice relativa aos termos da consulta, que torna a recuperação de informação muito mais eficiente. Por fim, com a repetição deste processo para todos os pares $(termId, docId)$, o arquivo onde as tuplas corretas foram escritas corresponderá ao arquivo invertido final.

Dessa maneira, com o Índice invertido criado, o programa armazena todo o vocabulário no arquivo "vocabulary.txt"e armazena a tabela hash com as URLs e o identificador dos seus respectivos documentos($docId$) no arquivo "doc_url.txt". Além disso, o programa armazena também as hashs que contém as posições onde cada um dos termos está armazenado nos índices. Por fim, o Indexador realiza o cálculo do PageRank e armazena o valor do pageRank de cada página no arquivo "pagerank.txt". Após gerar todos esses arquivos, o Indexador é finalizado.

3.2 Processador de consultas

Após a execução do Indexador e a criação do índice invertido, é possível realizar consultas em cima dos documentos da coleção. Para isso, foi criado o programa *Searcher*, que é composto pela classe principal *Searcher*

e pelas classes que representam cada um dos modelos de ranqueamento. Além disso, o programa é composto também pelo código do servidor, que executa o programa processador de consultas ao ser executado.

Dessa forma, o programa é iniciado com a execução do servidor no terminal, que coloca o servidor no ar na porta **8080**. Assim, o usuário pode realizar suas consultas na página HTML do processador de consultas, que pode ser acessada em `http://localhost:8080`. Porém, antes de subir o servidor, por meio do construtor da classe *Searcher*, o programa armazena em memória o conteúdo dos arquivos gerados pelo Indexador, com exceção dos arquivos invertidos. Por serem muito grandes, os dois arquivos invertidos(texto e *anchor texts*) são acessados em disco durante o processamento da consulta. É devido a esse acesso em disco que tornou-se necessário a criação dos arquivos contendo a posição dos termos em cada um dos índices. Dessa maneira, o programa acessa rapidamente o bloco do arquivo invertido relativo ao termo e a busca pode ser realizada com maior eficiência.

Após armazenar os arquivos em memória, o servidor estará no ar e as consultas podem ser realizadas. Ao realizar a sua consulta, o usuário pode selecionar um dos cinco modelos de ranqueamento para o processamento da consulta. Portanto, o programa pode realizar essa tarefa de cinco maneiras diferentes, que serão detalhadas a seguir. Por fim, depois de processar a consulta o programa identifica quais URLs correspondem aos IDs do resultado e retorna para o usuário as 20 URLs que correspondem aos 20 primeiros documentos do ranking gerado.

3.2.1 Modelo Booleano

Ao receber uma consulta, o modelo booleano separa as palavras da consulta e verifica no arquivo invertido a lista de documentos em que cada uma delas ocorre. É importante destacar que, assim como na indexação, stopwords e algumas outras palavras são ignoradas conforme os mesmos critérios da indexação. Assim, após obter o *set* com os IDs dos documentos em que ocorre cada palavra, o modelo simplesmente realiza a interseção entre os *sets* e retorna este conjunto final como resultado da busca.

3.2.2 Modelo Vetorial

No modelo Vetorial, por sua vez, as palavras da consulta também são separadas e é obtida a lista de documentos em que cada uma delas ocorre. Para coletar a lista de documentos em que esses termos ocorrem, o programa utiliza o valor do *seek* relativo ao termo e faz a leitura do bloco do arquivo invertido referente a este termo. Assim, o programa cria um vetor para armazenar a lista de documentos relativa a cada termo da consulta. Uma vez obtida essas listas, o programa calcula o $TF-IDF$ para cada termo. O TF é simplesmente o tamanho da lista de documentos em que o termo ocorre e o IDF também pode ser calculado facilmente, uma vez que se sabe o número de documentos da coleção. Depois, basta calcular a similaridade entre os vetores de cada documento e o vetor da consulta. Assim, o *score* final de cada documento corresponde à similaridade entre os vetores:

$$sim(d_j, q) = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \times \sqrt{\sum_{j=1}^{t} w_{i,q}^2}}$$

3.2.3 Modelo Vetorial combinado com Anchor Text

Este modelo tem implementação semelhante ao Modelo Vetorial. No entanto, a diferença está no índice invertido em que é obtida a lista de documentos em que o termo ocorre. Neste caso, o índice invertido utilizado é o índice dos *anchor texts*.

3.2.4 Modelo Vetorial combinado com PageRank

Este modelo também realiza o processamento da busca com o Modelo Vetorial. Porém, ao receber o resultado, o modelo adiciona ao *score* final o valor do PageRank da página. Devido ao fato de os valores do

PageRank serem bem menores que o *score* $V[DocId]$ gerado pelo Modelo Vetorial, este modelo realiza o seguinte cálculo do score de um documento:

$$Score[DocId] = V[DocId] + 10000 * PageRank[DocId]$$

3.2.5 Modelo Combinado

O Modelo Combinado tem a finalidade de gerar um resultado levando em consideração o Modelo Vetorial, os *anchor texts* e o PageRank da página. Assim, este modelo representa um combinação linear dos outros 3 modelos. Seja $V[DocId]$, PR[DocId] e A[DocId] os valores do *score* gerado pelos modelos Vetorial, Vetorial combinado com PageRank e Vetorial combinado com Anchor Text, respectivamente. O valor final gerado pelo Modelo Combinado pode ser representado pela fórmula:

$$Socre[DocId] = 0.85 * V[DocID] + 0.15 * A[DocId] + 10000 * PR[DocID]$$

4 Análise de Complexidade do Processador de Consultas

O processador de consultas é pode gerar os resultados utilizando 5 modelos diferentes: *Booleano, Vetorial, Vetorial combinado com PageRank, Vetorial combinado com AnchorText e Combinado*. Nesta seção, será feita a análise de complexidade para cada um desses métodos de ranqueamento. Para as análises a seguir, será utilizada a seguinte notação: T é o número de termos da consulta, N é o **número de páginas em que o termo ocorre** e A é o número de páginas que correspondem a links com *anchor text* onde o termo aparece.

- **Modelo Booleano** - Com o método booleano, o processador precisa verificar toda a lista de documentos em que cada termo ocorre. Ou seja, essa tarefa tem custo $\theta(N * T)$. Além disso, é necessário realizar a interseção entre as N listas de tamanho T, que também gera o custo $\theta(N * T)$. Portanto, este modelo tem custo médio $\theta(N * T)$.
- **Modelo vetorial** - Assim como o Booleano, o modelo Vetorial precisa verificar toda a lista de documentos em que cada termo ocorre e essa tarefa tem custo $\theta(N * T)$. Porém, depois de obter cada uma dessas listas, é necessário calcular o valor do TF e do IDF do termo. Como essa operação tem custo constante, a complexidade final do Modelo Vetorial é $\theta(N * T)$.
- **Modelo Vetorial combinado com PageRank** - Como o valor do PageRank de cada uma das páginas está armazenado em uma tabela hash na memória principal, o acesso tem custo $\theta(1)$. Assim, o custo desse processamento é o custo para processar a consulta com o modelo vetorial e, portanto, é $\theta(N * T)$
- **Modelo Vetorial combinado com Anchor Text** - Esse modelo temo custo semelhante ao Modelo Vetorial. No entanto, neste modelo a lista dos documentos tem tamanho A em vez de N. Portanto, o custo final é $\theta(A * T)$.
- **Modelo Combinado** - Como o valor do PageRank de cada uma das páginas está armazenado em uma tabela hash na memória principal, o acesso tem custo $O(1)$. Assim, o custo desse processamento é o mesmo custo para processar a consulta com o modelo vetorial somado ao custo do modelo Vetorial combinado apenas com o Anchor Text e, portanto, é $\theta(N * T + A * T)$ ou $\theta((N + A) * T)$.

5 Análise Experimental

5.1 Metodologia

Com o objetivo de fazer uma análise dos resultados da busca realizada, neste trabalho foram realizados alguns experimentos. O primeiro experimento apresenta os dados estatísticos do processo de indexação da coleção. O segundo, por sua vez, realiza uma avaliação experimental da análise de complexidade feita na seção anterior e verifica a relação entre a quantidade de memória disponível e o tempo de execução do programa.

A implementação dos experimentos foi feita com a utilização do compilador G++, utilizando a flag $-std = c + +11$. Os testes foram executados com o uso de HD externo (5400 rpm) em uma máquina com 8GB de memória RAM, processador Intel i5 2.7 GHz e sistema operacional OS X Sierra.

5.2 Resultado dos experimentos

Neste trabalho, foi realizada a indexação de uma coleção de páginas em HTML coletada da Web brasileira. Com o término da execução do Indexador, foi gerada a tabela 1 com os dados do índice invertido e da coleção em geral.

Tabela 1: Dados Estatísticos da Indexação

Número de Documentos	6.261.116
Número de termos do Vocabulário	9.766.028
Tamanho Índice Invertido	108,33 GB
Tamanho Índice Invertido - Anchor Text	1,89 GB
Tempo total de indexação	46h 29 min

Com o objetivo de avaliar os diferentes modelos de ranqueamento utilizados pelo programa, foram selecionadas cinco consultas para serem testadas: “itau”, “donald trump”, “como fazer crepioca”, “o que é filosofia”e “por que o feijão está tão caro”. Para cada consulta, foi realizada a busca utilizando cada um dos modelos e gerou-se uma tabela para que se possa visualizar o grau de relevância do documento que apareceu em cada uma das cinco primeiras posições do ranking. Esse grau de relevância tem valor entre 0 e 5, conforme o critério a seguir:

- **5** - URL exata referente a consulta;
- **4** - Não é a página exata, mas leva à exata;
- **3** - Informação de interesse pode (might) estar na página (exigiria maior inspeção);
- **2** - Até está relacionado, mas não é primeira ou segundo melhor resultado;
- **1** - Não é relevante ou não está relacionado;
- **0** - Página não encontrada (fora do ar ou link no apontando para uma página inexistente);

Para cada uma dessas consultas, foi fornecido uma lista de documentos e a sua respectiva relevância para a consulta. Assim, os documentos que eventualmente apareceram no ranking e não estavam presente nessa lista foram avaliados manualmente para se verificar a relevância do documento.

Tabela 2: Grau de Relevância da Consulta Itau

Ranking	Consulta	Combinado	Vetorial	Vetorial + PageRank	Vetorial + Anchor Text
1	itau	1	4	4	1
2		4	1	1	5
3		1	1	1	1
4		2	2	2	1
5		2	2	2	3

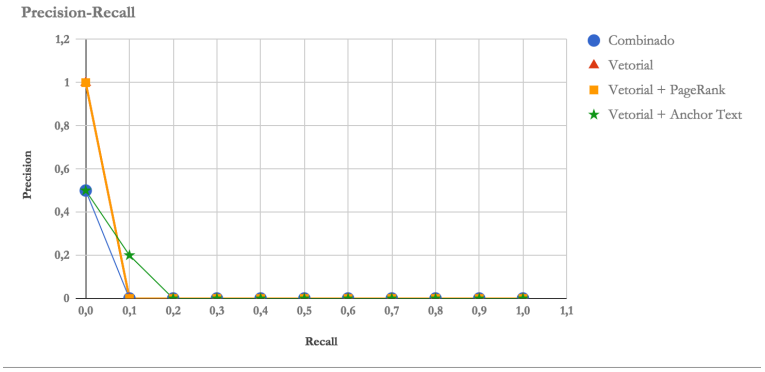

Figura 3: Precision-Recall para consulta itau

Para a consulta "Itau", os modelos vetorial e vetorial combinado com pageRank apresentaram resultado semelhante e conseguiram recuperar um documento relevante na primeira posição do ranking. Apesar de ter colocado um documento não relevante na primeira posição, pode-se dizer que o modelo vetorial combinado com *anchor text* atingiu resultado um pouco melhor, já que foi o único modelo que recuperou um documento totalmente relevante e o único que apresentou duas páginas relevantes entra as cinco primeiras.

Portanto, pode-se dizer que considerar os *anchor texts* nesta consulta colaborou com a melhora do resultado. Uma possível justificativa para essa melhora é o fato de "Itau"ser uma consulta simples e bem específica que espera como resultado a página principal do Banco Itaú. Como a maioria das referências a esta página provavelmente tem a palavra "Itau"como parte do a*nchor text*, o modelo combinado com *anchor text* agrega bastante valor ao resultado da consulta. Possivelmente, outras consultas parecidas, como "vivo"e "Santander"por exemplo, terão resultado semelhante.

Tabela 3: Grau de Relevância da Consulta Donald Trump

Ranking	Consulta	Combinado	Vetorial	Vetorial + PageRank	Vetorial + Anchor Text
1	Donald Trump	3	0	0	3
2		1	1	1	3
3		3	1	1	3
4		0	1	1	3
5		4	1	1	3

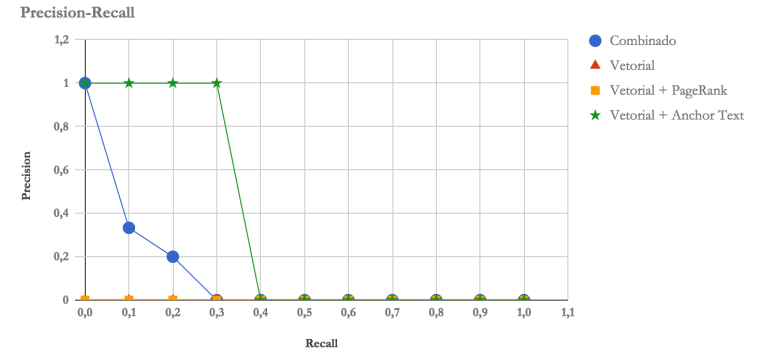

Figura 4: Precision-Recall para consulta Donald Trump

Já em relação à consulta "Donald trump", o resultado só foi bom para os modelos que consideraram os *anchor text* para a geração do ranking. Neste caso, o fato de o termo "Donald Trump"aparecer em diversas páginas e em diferentes contextos atrapalha bastante o modelo vetorial. Outro fato que contribui para o fracasso do modelo vetorial é o fato de a coleção ter sido coletada anteriormente e o termo "Donald Trump"ter sua maior presença em sites de notícias. Assim, a maioria dos sites de notícias que continham alguma notícia relevante sobre Trump no dia da coleta, podiam não ter mais a notícia em sua página na hora da avaliação.

Tabela 4: Grau de Relevância da Consulta O que é filosofia

Ranking	Consulta	Combinado	Vetorial	Vetorial + PageRank	Vetorial + Anchor Text
1	o que é filosofia	5	4	4	1
2		2	1	1	1
3		5	5	5	2
4		4	1	1	1
5		3	1	1	1

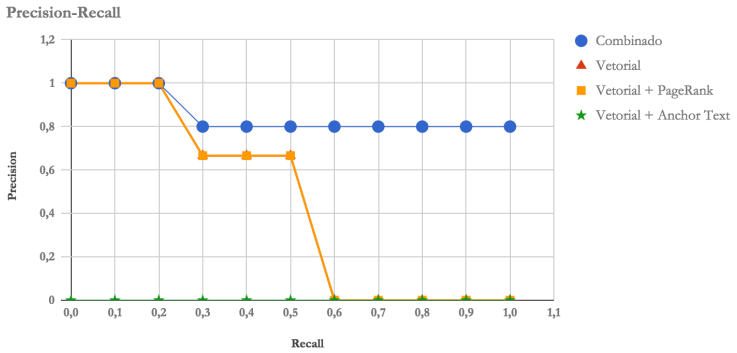

Figura 5: Precision-Recall para consulta O que é Filosofia

Apesar de ser bem específica, a consulta "O que é filosofia"acaba se tornando bem genérica nesta máquina de busca, devido ao descarte das stopwords. Neste caso, a busca considerada pelo processador de consultas foi apenas "Filosofia". Assim, devido à generalidade da consulta, o modelo combinado com anchor text teve péssimo resultado. Por outro lado, os outros modelos que dão maior peso ao modelo vetorial simples tiveram resultados bons, já que a simples presença frequente do termo "filosofia"nas páginas já indica um possível resultado relevante para a busca.

Tabela 5: Grau de Relevância da Consulta Como fazer crepioca

Ranking	Consulta	Combinado	Vetorial	Vetorial + PageRank	Vetorial + Anchor Text
1	como fazer crepioca	5	3	3	1
2		5	5	5	1
3		3	5	5	1
4		5	0	0	1
5		5	5	5	1

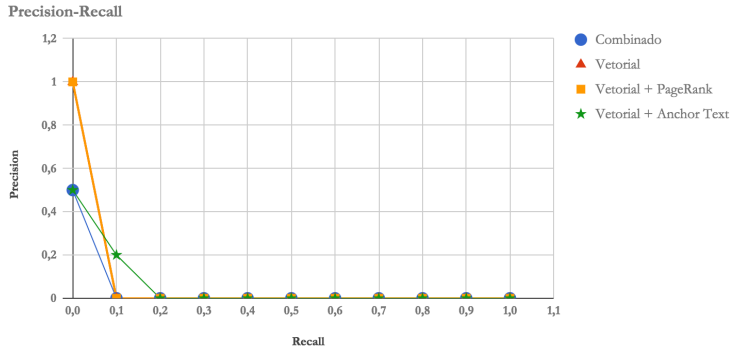

Figura 6: Precision-Recall para consulta Como Fazer Crepioca

A consulta "Como fazer crepioca"também é bem específica. Além disso, a simples presença do termo "Crepioca"indica que a página tem grandes chances de ser relevante. Provavelmente, esses são os motivos dos

bons resultados das buscas que levaram em consideração o modelo vetorial. Por outro lado, "crepioca"não deve ser um texto muito utilizado em *anchor texts* e, por isso, torna o *anchorRank* tão ruim para essa consulta.

Tabela 6: Grau de Relevância da Consulta Por que o feijão esta tão caro

Ranking	Consulta	Combinado	Vetorial	Vetorial + PageRank	Vetorial + Anchor Text
1	por que o feijão está tão caro	1	1	1	1
2		1	1	1	1
3		1	1	1	1
4		1	1	1	1
5		1	1	1	1

Por fim, a consulta "Por que o feijão está tão caro"apresentou um péssimo resultado em todas os modelos de ranking. Provavelmente, o motivo deste resultado ruim foi a complexidade da consulta agregada ao pequeno número de páginas relevantes na coleção. Para esta consulta, além do processador de consultas não ter recuperado documentos relevantes, a lista fornecida com a relevância das URLs também não apresenta nenhum resultado relevante.

Devido ao fraco comportamento do ranking gerado na maioria das consultas, obter as curvas de Precision and Recall pode não ser a melhor métrica para avaliar as consultas. Neste caso, utilizar uma métrica com valor absoluto pode ser mais viável. Para isso, foi gerada a **Tabela 7**, onde é possível visualizar o valor de *P@5* para cada uma das consultas do experimento.

Tabela 7: Valores de P@5 para as consultas

Consulta	Combinado	Vetorial	Vetorial + PageRank	Vetorial + Anchor Text
Itau	0,2	0,2	0,2	0,2
Donald Trump	0,6	0,0	0,0	1,0
Como Fazer Crepioca	1,0	0,8	0,8	0,0
O que é filosofia	0,8	0,4	0,4	0,0
Porque o feijão está tão caro	0,0	0,0	0,0	0,0

A partir dos resultados, é possível notar que o comportamento de um modelo pode variar em relação à consulta. Na consulta "Como fazer Crepioca"por exemplo, o modelo que apresenta melhor resultado é o modelo combinado. Por outro lado, o modelo com anchor text foi o melhor na consulta "Donald Trump", quando este modelo retornou somente páginas relevantes. Assim, é possível concluir que cada modelo de ranqueamento tem suas vantagens e desvantagens e, portanto, podem apresentar resultados bons ou ruins, dependendo da consulta. Dessa forma, assim como mostra a **Tabela 7**, a melhor opção na maioria dos casos é gerar um modelo combinado, que leva em consideração diferentes métodos.

6 Conclusão

O Trabalho Prático 3 teve como objetivo a implementação de um Processador de consultas em uma coleção estática com mais de 6 milhões de páginas em HTML coletadas da Web brasileira. Para realizar a busca, o programa podia utilizar 5 métodos distintos para calcular o *score* que determinava o ranking final dos documentos recuperados. Para avaliar esses modelos de ranqueamento, foram realizados experimentos utilizando métricas de avaliação como *Precision* e *Recall*.

Por fim, pode-se declarar que o objetivo do trabalho foi cumprido com sucesso e que este teve importância fundamental para o aprendizado e e para o entendimento concreto do funcionamento de um processador de consultas, assim como o funcionamento de cada uma das etapas necessárias para a construção de uma máquina de busca.